Scalable data format for Step datatable

Goals

To investigate and depend on data format that is suitable for Riffyn data and supports large datasets. The data format should have the following capabilities

1. Lightweight
2. Multi-threaded support (for chunking if need be and distribution)
3. Querying (We should be able to query the data format fairly easily - SQL like)
4. client library available in multiple languages (JS, Python - backend code in JS needs to read datatables for clean-tab/)

Formats under consideration

*These are all the formats that are being tested now, new formats will be added and investigated on on suggestions or further research

1. Apache parquet (Compressed and uncompressed)
2. gZip
3. HDF5
4. CSV
5. Parquet
6. parquet-brotli
7. parquet-snappy

Measurements

All measurements are taken on a machine with the following specs:

Processor Name: Intel Core i7

Processor Speed: 2.2 GHz

Number of Processors: 1

Total Number of Cores: 4

L2 Cache (per Core): 256 KB

L3 Cache: 6 MB

Memory: 16 GB

The file formats under consideration now are different compressed versions of

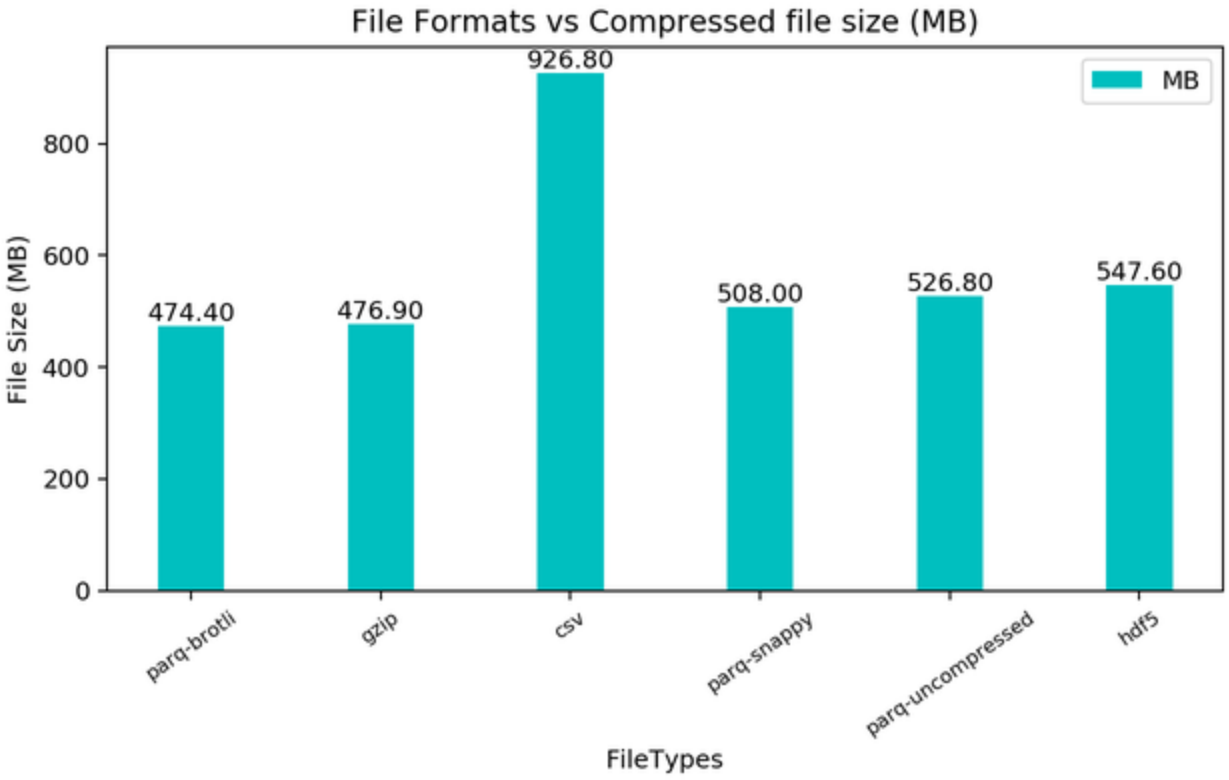
1. Parquet
2. HDF5
3. CSV
4. gZip
5. *Apache Avro* looks promising but haven't been tested. Will be brought in if we ever use some form of "Streaming".

Performance metrics

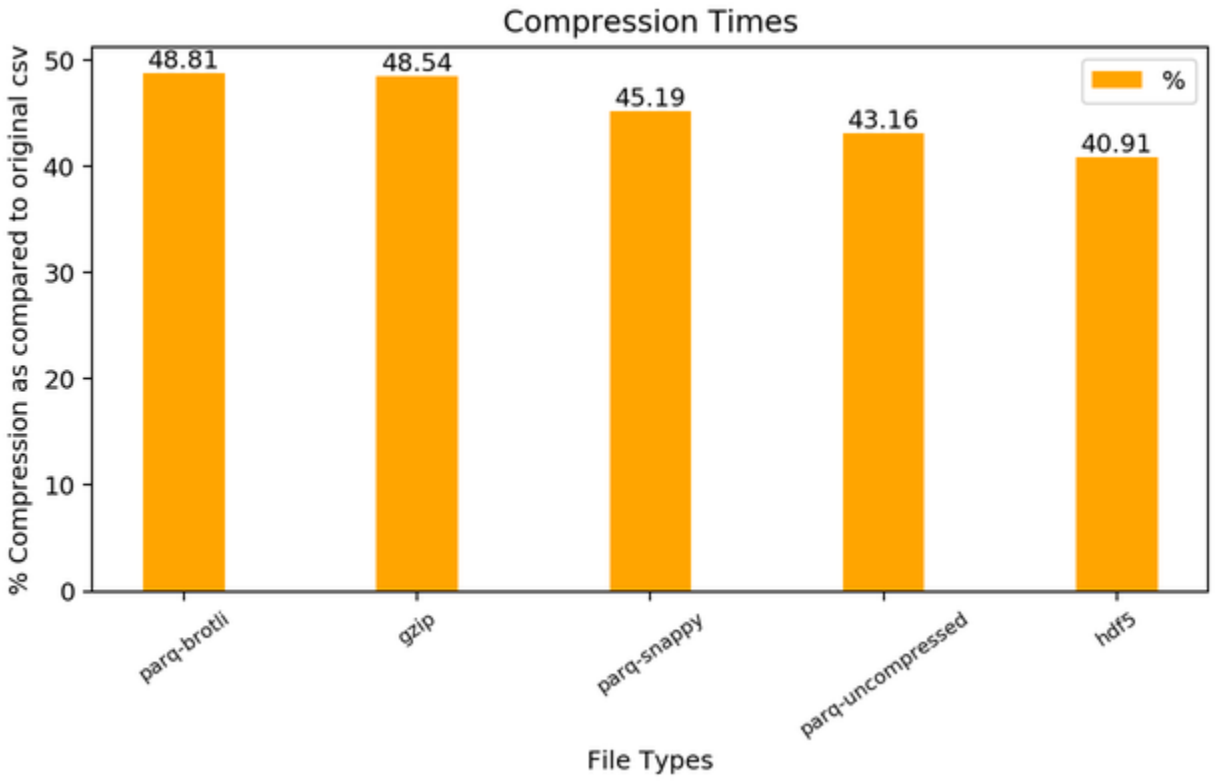
Performance metrics that would define the quality of the data format on single threaded / multi-threaded environments would be.

1. Compression ratio
2. Read Time
3. Compression Time
4. Uncompress Time

Compression Ratio : For a CSV file size of 926.8 MB, the % compression for different file formats listed and the actual file sizes are as shown.



Compression Times

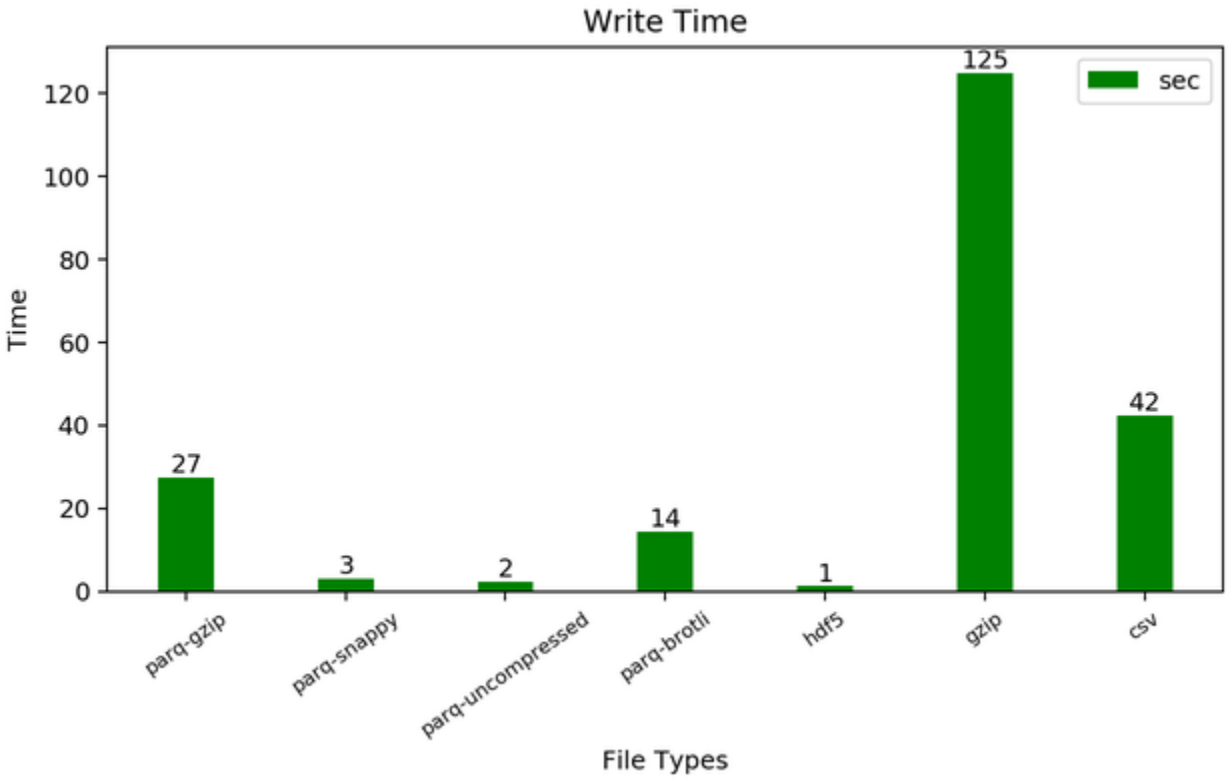


Write Time

The writing time of the compressed files also determines the suitability.

#gzip= CPU times: user 25.5 s, sys: 2.01 s, total: 27.5 s Wall time: 26.1 s
#snappy = CPU times: user 2.84 s, sys: 240 ms, total: 3.08 sWall time: 2.8 s
#uncompressed = CPU times: user 2.12 s, sys: 241 ms, total: 2.36 sWall time: 2.06 s
#Brotli = CPU times: user 12.1 s, sys: 2.3 s, total: 14.4 s Wall time: 14.1 s
#hdf5 = CPU times: user 455 ms, sys: 757 ms, total: 1.21 sWall time: 1.81 s
#Direct gzip = CPU times: user 3min 45s, sys: 2.02 s, total: 3min 47sWall time: 3min 49s

Format	CPU Time (in seconds)		
	User	**System**	**Total**
uncompressed	2.12	0.02	2.36
gzip	25.50	2.01	27.50
snappy	2.84	0.02	3.08
Brotli	12.10	2.30	14.40
hdf5	0.45	0.75	1.21
Direct gzip	225.00	2.02	227.02

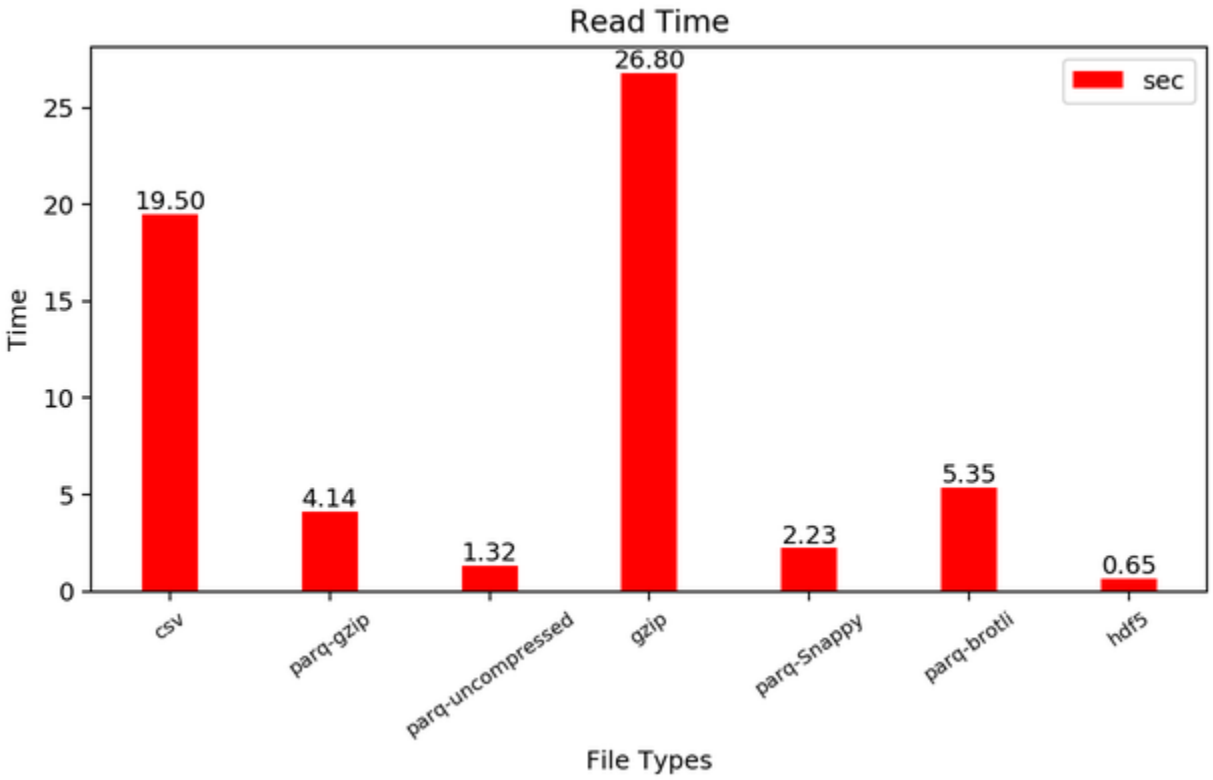


Read Times

The read times are a function of how fast pandas read the data because we still want to read it as a Pandas data-frame until we have another solution (Parquet?)

Read Times
Direct csv = CPU times: user 18.1 s, sys: 1.36 s, total: 19.5 s Wall time: 19.9 s
Parq-gzip = CPU times: user 4.14 s, sys: 1.16 s, total: 5.29 s Wall time: 6.89 s
Parq-uncompressed = CPU times: user 1.32 s, sys: 1.07 s, total: 2.39 s Wall time: 3.79 s
gzip = CPU times: user 25 s, sys: 1.78 s, total: 26.8 s Wall time: 27 s
hdf5 = CPU times: user 125 ms, sys: 570 ms, total: 695 ms Wall time: 1.71 s
fastparquet gzip multithreaded = CPU times: user 3.4 s, sys: 455 ms, total: 3.85 s Wall time: 3.86 s
fastp gzip single = CPU times: user 3.41 s, sys: 470 ms, total: 3.88 s Wall time: 3.89 s
parquet + snappy = CPU times: user 1.4 s, sys: 833 ms, total: 2.23 s Wall time: 1.77 s
parq- BROTLI = CPU times: user 4.56 s, sys: 783 ms, total: 5.35 s Wall time: 5.01 s

Format	CPU Time (in seconds)		
	User	**System**	**Total**
CSV	18.10	1.36	19.50
parquet + gzip	4.41	1.16	5.29
parquet + uncompressed	1.32	1.07	2.39
parquet + snappy	1.40	0.83	2.23
parquet + BROTLI	4.56	0.78	5.35
	225.00	2.02	227.02
fast-parquet + gzip + single-threaded			
fast gzip + single			



Advantages/ Disadvantages

We can see based on the metrics that hdf5 has the fastest read and write times and also has a good compression ratio. The compression ratio is highest in all cases in Gzip but Gzip doesn't allow Multi-threaded read or write because of the way it is compressed. The most popular multi-threaded, fast read and write support is offered in Parquet. Although the Compression to be used is still debatable, Parquet will be used for now with Google's Snappy Compression, for two reasons,

1. Multi-threaded support
2. Efficient Querying (We can query directly on compressed Parquet file)
3. Large Open source community that has support with multiple languages (Apache arrow Project, Spark Parquet libraries)

For Future: Other format that has all these features is ORC. Less overall supporting libraries is the reason why it has been not used for *Riffyn* data tables

The downloadable CSV file will be saved as gzip since export is always triggered at the end but also at the expense of Time of execution or compression Time (which is more than any other kind of compression).

Components

The minimum components that need to be created / changed for supporting the new data format are

1. Read / Write Datatables directly to S3 or any intermediate storage system
2. Querying support on the datatables (prefereably in thier raw format without having to convert them to any other kind of format in memory just for the sake of querying on the tables)
3. Joining. All data joinings whether it is intrarun / interrun / inter-activity / interexperiment (which is the process level datatable) has always been performed using the traditional pandas sql style joins. Much efficient and preferable style of joining would support joining on the raw format without having to load it into memory as pandas (or some ohter kind of dataframes) and also should efficient in terms of speed of joining the datatables.

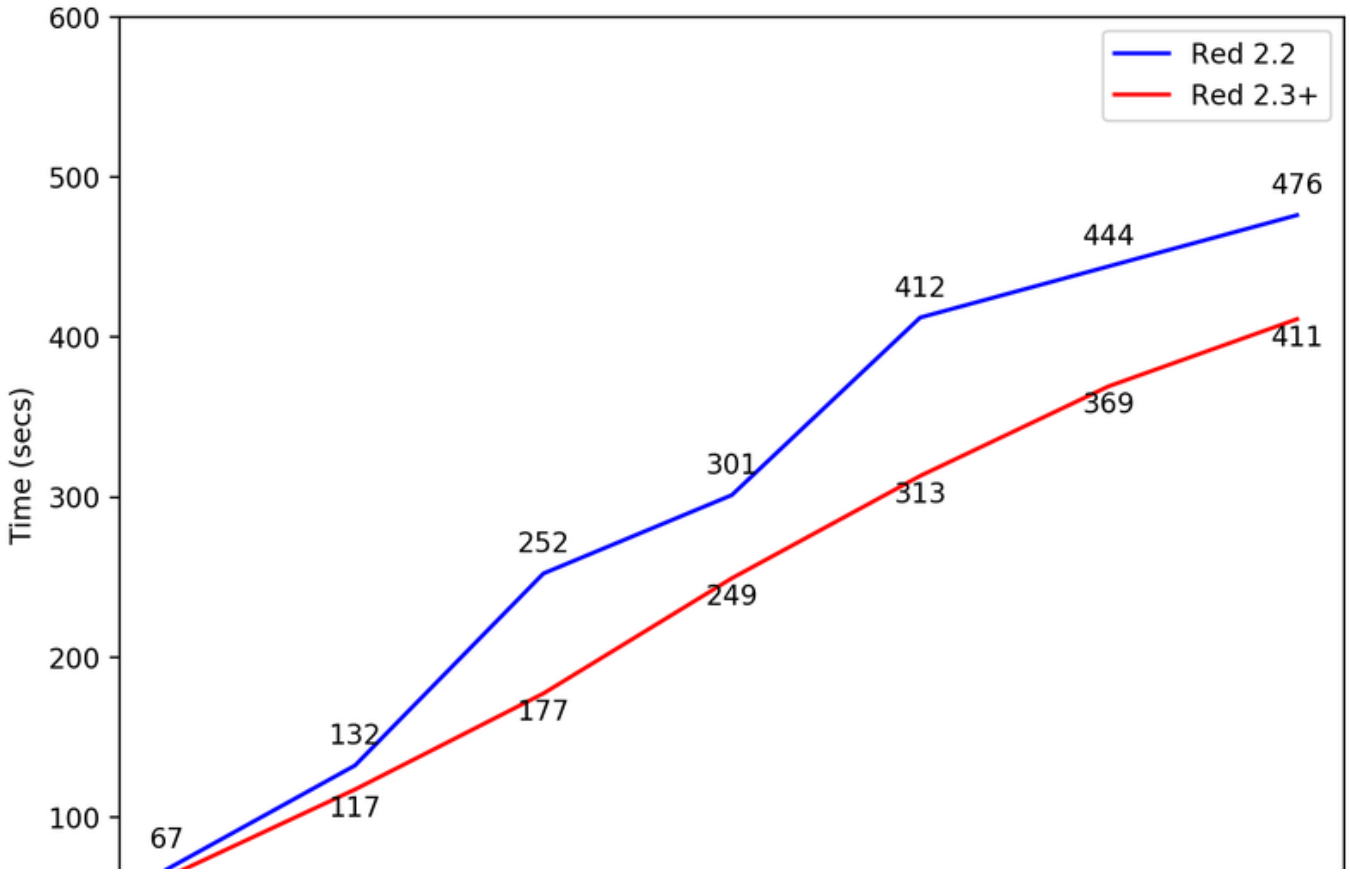
Phase-1

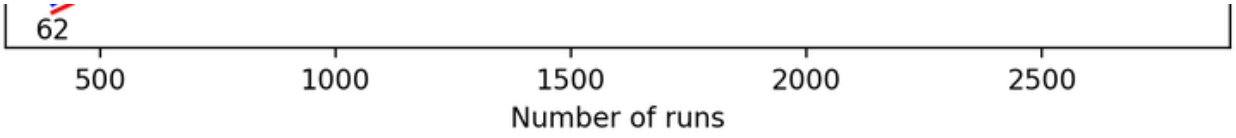
phase 1 has the following completed characteristics

1. Interface to S3 for read. handles read/write in different (Parquet (Snappy) and CSV as of RED 2.2) formats
2. Writes activity level to s3 key with the hierarchy as DatatablesCollection/Datatables/top_group_id/expt_id/activity_id_cleanprocId
3. Experiment level data tables are written to key DatatablesCollection/Datatables/top_group_id/expt_id/export_config_id
4. All internal data is written to S3 as parquet. No more CSV. The only time we write to CSV is the expt data table when export data is triggered.
5. Migrate existing fresh experiment datatable to avoid data-ingestion issue for customers' data-lake. Is it really required?

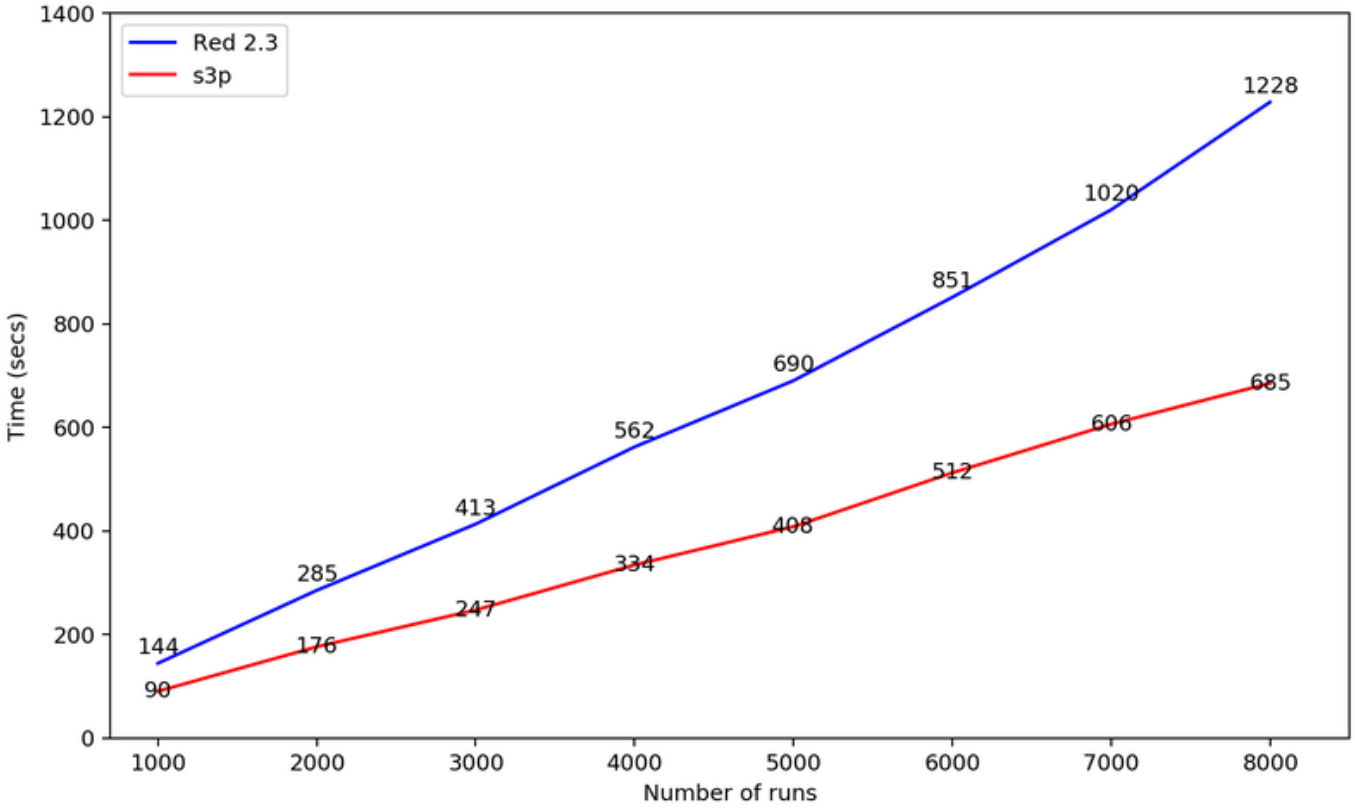
PERFORMANCE:

June 18, 2018



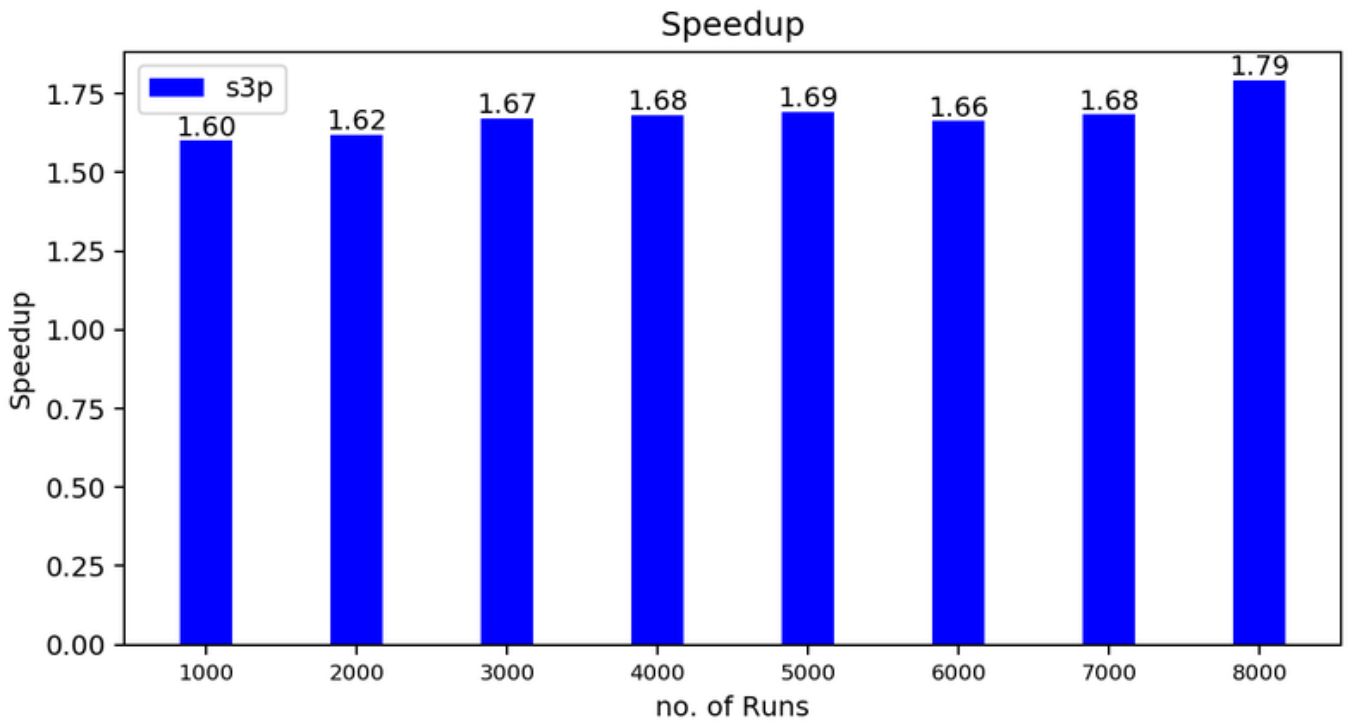


JULY 2018



SPEEDUP

JULY 2018



Phase-2

1. Abstract the data table in a meta store such as HIVE. (Refer to this page: Meta store and SQL support)
2. In-place update data tables (Possibly Stream continuously to the Storage (may not be s3)
3. When meta storage layer is abstracted, use it to query on the data
4. Build complete ETL system for analytics

Query Support Components (WIP) (Refer to this page: Meta store and SQL support)

1. Engine
2. Storage amazon s3
3. meta storage Postgres
4. Compute Engine RED
5. JDBC/ODBC Drivers